NASA TM-103,289

NASA Technical Memorandum 103289

NASA-TM-103289 19910002902

Shared Direct Memory Access on the Explorer II–LX

Jeffrey L. Musgrave
Lewis Research Center
Cleveland, Ohio

September 1990

LIBRARY COPY

NOV 19 1990

LANGLEY RESEARCH CENTER
LIBRARY, NASA
HAMPTON, VIRGINIA

NASA

Table of Contents

N91-12215 #

I. INTRODUCTION

In the area of intelligent control systems, typical problems require an integrated computing environment for handling the special processing needs associated with the various levels of a control hierarchy. Lower levels which interact directly with the process require high degrees of accuracy and speed while higher levels which interact with subsystems in the control hierarchy are more decision oriented and better represented in a symbolic processing environment. This work is directed toward providing enhanced computational capabilities and exploiting the special features of the Texas Instruments (TI) Explorer II-LX for rapid prototyping of intelligent control systems.

The TI Explorer II-LX is a unique machine having two microprocessors contained in a single chassis. One microprocessor is a Lisp chip where Lisp, a commonly accepted language for AI applications, is implemented in hardware. The other is a M68020 which is the first full 32-bit implementation of the M68000 family. The operating system on the Explorer is a Common Lisp Interpreter, and System V Unix runs on the M68020. The goal is to make this powerful feature more accessible to the user without requiring advanced knowledge of the low level details associated with shared memory.

A. System Level Requirements

The following is a list of hardware and software requirements for utilizing the shared memory feature on the TI Explorer II-LX system:

1. TI Explorer II-LX,
2. Explorer Software Version 4.0 or Higher,
3. System V Unix Version 2.2 or Higher,
4. C Compiler and Fortran Compiler (optional),
5. LX Software Version 3.0 or Higher.

B. Software Requirements

1. Integration in terms of data sharing between computationally oriented programming languages and those more suited to the implementation of "intelligent" or higher level functions which are decision oriented.

2. Simple methodology with a generic structure for relative ease of use and maximum flexibility.

3. Streamlined approach for high performance in terms of raw speed to maximize benefits associated with concurrent processing.

4. Synchronous and asynchronous processing capability with assured data integrity.

C. Approach

Shared direct memory access (SDMA) is the direct sharing of information through memory common to each of the microprocessors on the TI Explorer II-LX. In particular, the setup of the shared memory and an example are given in Chapter 5 of [1]. The goal here is to design a suitable interface for using the shared memory capability effectively without requiring explicit knowledge of the low level details.

D. Features and Limitations

Before using SDMA, the programmer should make a careful assessment of his computational requirements and needs with respect to the user interface in both the short and long term. Clearly, an SDMA solution will be more complex in design and implementation and consequently more difficult to debug and verify than more traditional software solutions. Additional burden will be placed on the programmer since he must be well versed in a variety of languages with a variety of programming styles.

In order to assist the programmer in assessing the potential costs and benefits of an SDMA solution for his particular problem, the following provides an outline of some important topics for consideration.

Features

Integrated environment for numerical and symbolic processing.

Synchronous or asynchronous processing capability.

Performance enhancements resulting from two processors working in parallel or tandem with solutions in the domain most suited to the problem description and structure.

Window driven user interface on the Lisp side complete with menus and mouse.

Object oriented style of programming in Lisp for rapid design and enhanced reusability of code.

Capability for treating the M68020 as a slave processor by the Lisp side resulting in stream-lined applications with a single driver.

Limitations

Complex solution involving different software environments with a parallel architecture.

Difficult design since applications must conform to the interface established by the SDMA primitives.

Only single blocks of shared memory are currently supported requiring all shared information to pass through a single data channel.

Data buffering is not supported on either the Lisp side or the Unix side. If the application requires this capability, then the programmer is responsible for setting up the buffer and time stamping the data. However, this can be accomplished in the current framework without modification to the primitives.

Applications are more difficult to debug since an SDMA solution requires a parallel architecture and proper usage of the primitives.

E. Example of Memory Sharing

In order to motivate the shared memory approach and provide a clearer picture of what is meant by shared memory, a simple example is given which makes use of the shared memory capability.

Let $f()$ represent a Fortran program with input/output behavior of the form

$$\mathbf{x}_{i+1} = f(\mathbf{x}_i, \mathbf{u}_i)$$

where $\mathbf{x} \in \Re^N$ and $\mathbf{u} \in \Re^M$. In a similar fashion, let g() represent a Lisp program with input/output behavior of the form

$$\mathbf{u}_i = g(\mathbf{x}_i).$$

Given the above data requirements, the vector $\mathbf{x}_i$ must be passed from Fortran to Lisp where the vector $\mathbf{u}_i$ is generated. Vector $\mathbf{u}_i$ must then be given back to the Fortran program in order to perform a major iteration (ie. generate $\mathbf{x}_{i+1}$). Given the fact that disk access is typically the

slowest operation performed by a computer, a file sharing approach would make the above scenario impractical for most applications since data must be passed twice for each major iteration. However, shared memory may be utilized to dramatically enhance performance without converting **g** to Fortran or f to Lisp.

Using the above notation, a shared memory approach as proposed in this work would take the following form:

```
i = 0
while Not Finished do

        Send x_i to shared memory from Fortran;
        Read x_i from memory and give to the waiting Lisp function
                g and generate u_i;
        Send u_i to shared memory from Lisp;
        Read u_i from memory and give to the waiting Fortran function
                f to generate x_{i+1};
        Set i to i + 1;
        Check stopping criterion.
```

In this example, parallelism cannot be fully exploited due to the information processing requirements imposed by the problem structure. However, benefit is achieved by allowing two different computer programs running on separate microprocessors to pass information via common memory.

II. Shared Memory

The Explorer and the M68020 based machine are virtual memory systems. In other words, physical memory is not accessed directly by programs. Programs refer to virtual memory addresses which reside in physical memory (ie. RAM) or on disk. The memory used here will reside in physical memory in order to reduce access time.

A block of memory in a computer may be visualized as in Fig. 1. In general, the data structure corresponding to the physical memory can be of any design. However, it is simplest to visualize the physical memory as an array of contiguous data blocks containing an address and the data associated with the address as shown. Consequently, data can be shared between the two processors by setting up a block of physical memory and providing each with the addresses of all allocated memory cells. Once the address of physical memory is known, either processor can read from it or write to it just like any other cell in memory without regard to where the physical memory actually resides. This is the basic advantage of SDMA since direct memory access is the fastest way to store and retrieve information and the details with respect to the memory configuration are transparent to the application.

Fig. 2 shows possible paths of data flow under the current architecture. It should be clear from the figure that a great deal of flexibility exists in the utilization of information and the partitioning of the task based on the strengths and weaknesses of each microprocessor. Hence, it is advisable to use the structure inherent in the problem description as a guide to determining the interaction between the subtasks which constitute a proposed solution. A streamlined design with a configuration based on the structure of the problem will run faster and be much simpler to debug than an equivalent design where the structure is imposed arbitrarily. Additional complexity results from the fact that parallel architectures are relatively new and many programmers are unfamiliar with the common pitfalls and how they can best be avoided.

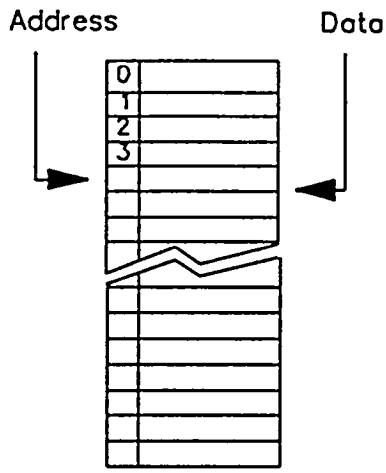

Fig. 1 Typical Block of Shared Memory

In the current architecture, the block (multiple blocks are not supported in the present design) of memory used for sharing information resides on the Unix side. Hence, the Lisp machine must have access to the 68020-based processor board's on-board memory (termed S1500 memory). In order to give the basic idea without getting bogged down in the details (see Appendix), the following overview is given as found in [1]:

Step:

1. Allocate blocks of S1500 virtual memory for sharing with the Lisp processor.

2. Page the blocks in. (set up the virtual memory)

3. Lock the pages so they are not paged out to disk while being used by an application.

4. Send the Lisp processor a memory map that indicates physical memory addresses corresponding to the locked virtual memory addresses.

5. Unlock the pages for reuse once the application has finished.

A. Shared Memory from Lisp

Once Lisp has been provided with a memory map of the wired down memory on the M68020 processor board, access may be achieved by directly indexing the contiguous cells of

storage. TI has supplied several low-level Lisp functions for performing this operation as listed in [1]. In other words, Lisp cannot access the memory cells directly since they are not resident on the Lisp processor board. Lisp must send data out or receive data via supplied procedures which index the contiguous cells of memory. In order to perform this operation, the type of information (eg. floating point, signed integer, etc.) and the number of bits (eg. 16, 32, 64) must be consistent with the data specifications set forth on the Unix side. In order to avoid type conflicts, the default data representation is 32 bit floating point since this is the most general data type for typical engineering applications. Hence, all data sent out by Lisp is converted to this format before being placed in shared memory. This operation is transparent to the user but should be kept in mind when designing code.

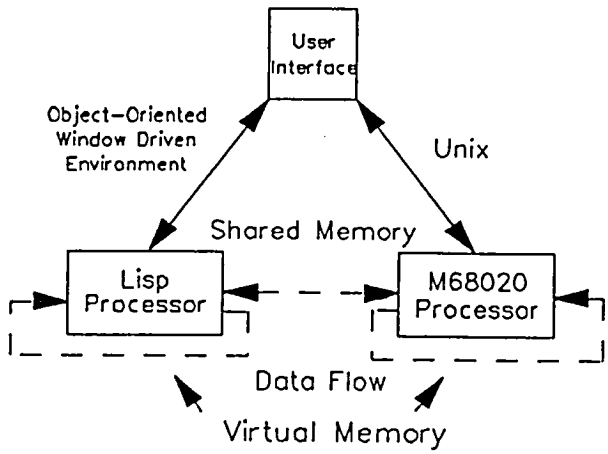

Fig. 2 Paths of Data Flow

In order for data to be sent or received, the number of elements to be shared at any instant must be known in order for all the data to be read. For simplicity, the maximum amount of data to be shared on any given cycle is written out to memory. However, this approach may be very costly in terms of overall performance if the application demands a high variation in the amount information to be shared on consecutive cycles. In the present setting (ie. Intelligent Controls), the variation in the amount of shared information will be small since the manner in which the data is used should not change as a function of time thereby providing justification for the approach taken here.

B. Shared Memory from Unix

Shared memory is allocated from the M68020 processor-based memory. Hence, the size of the memory and the type of data contained therein are defined by the Unix based application in a manner consistent with the information sharing requirements established for the task. On the Unix side, the shared memory is defined as an array of 32 bit floating point numbers of length MAX_DATA, where MAX_DATA is a prespecified constant (default value is 100). The maximum amount of shared information on a given cycle is bounded by MAX_DATA - 3. Three cells of shared memory are required for administrative purposes as shown in Fig. 3. The purpose for the reserved cells in the shared memory block will be discussed in detail below.

Since the interface with shared memory is an array, care must be taken when accessing the information since arrays are passed by reference as opposed to value in most general purpose high level programming languages (eg. Fortran). In other words, reassigning elements in the array immediately alters the contents in shared memory. A common hazard is the contamination of the data or the control flag used to describe the current state of memory

by the Lisp routines before the Unix side has finished reading the data. In order to avoid this event, a temporary buffer is used to hold the data until it is needed for processing while simultaneously releasing the Lisp based application to continue processing. The Lisp side need not be concerned with such details since access to shared memory is achieved via a procedure call which forces usage of a temporary buffer for direct access of information since Lisp functions pass all parameters by value. This is the most fundamental difference between the Unix side and the Lisp side with respect to usage of the common memory and should be kept in mind when using the primitives.

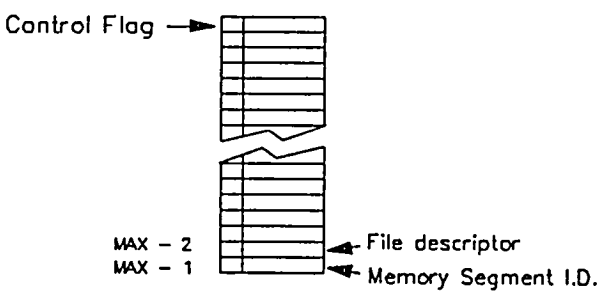

Fig. 3 Reserved Memory Cells

C. Administration of Memory

The total number of floating point numbers which may be sent or received is determined by the Unix side since the shared memory resides on the M68020 board. The total number of memory cells has a default value of 100. However, not all of the cells can be used for data sharing since three of them are required for administrative purposes as shown in Fig. 3.

The first cell (control flag) in the shared memory block is used to specify the state of the block. In particular, two states are needed for the handshaking associated with any of the described modes of operation. The first state, denoted by MEM_READ, is an indication that the memory contains newly written information. The second state, denoted by DONE_READING, is an indication that the newly written information has been read by the process requiring the data (see Figs. 4 - 6).

The last two blocks are descriptors which are needed to release the locked down memory to the heap after the sharing process has terminated. If these are corrupted, the memory cannot be released and an error message will be given. If a large amount of shared memory is required (ie. larger than 97 cells) then the default value must be changed to accommodate the information. However, it is very important that a **sufficient** amount of memory is allocated in order to avoid contamination of the descriptors. If more memory is requested (via user specified input) than has been allocated, an error will be given and the process aborted. Hence, the advantage of sending the same number of blocks on every cycle based on some a priori calculation is the guarantee of sufficient memory to share the necessary data before the application begins.

D. Modes of Operation

Associated with the shared memory are three basic modes of operation.

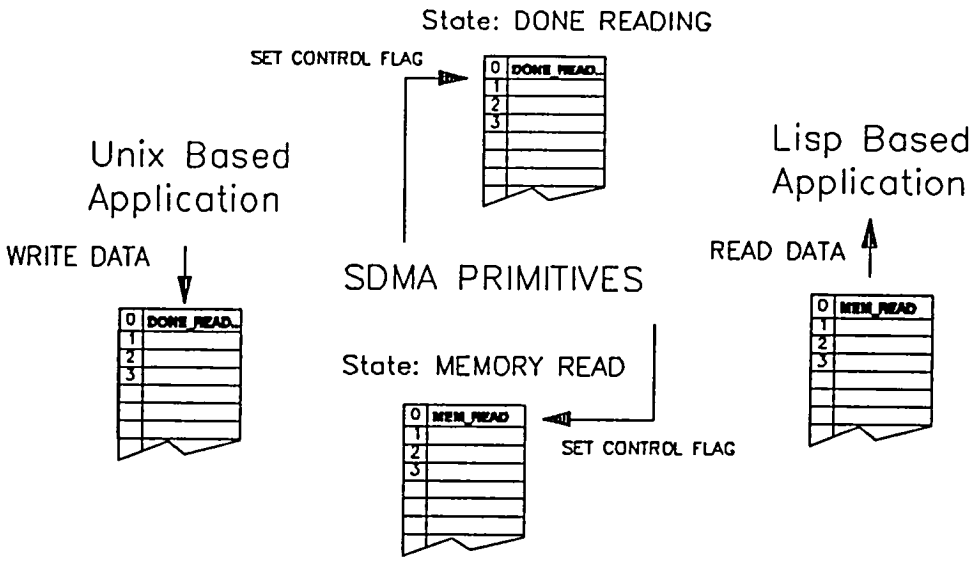

Fig. 4 Read Mode Cycle

Read Mode:

A Lisp based application receives information from an independently operating Unix based application as shown in Fig. 4. The Unix based application is responsible for terminating the shared memory connection by sending a FINISHED message to the memory driver when all data have been sent.

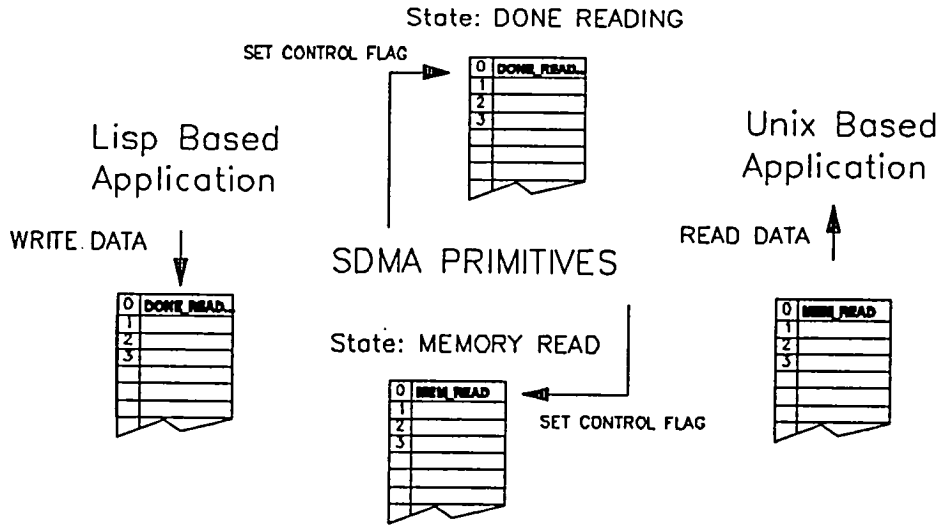

Fig. 5 Write Mode Cycle

Write Mode:

A Unix based application receives information from an independently operating Lisp based application as shown in Fig. 5. The Lisp based application is responsible for terminating the shared memory connection by sending a FINISHED message to the memory driver when all the data have been sent.

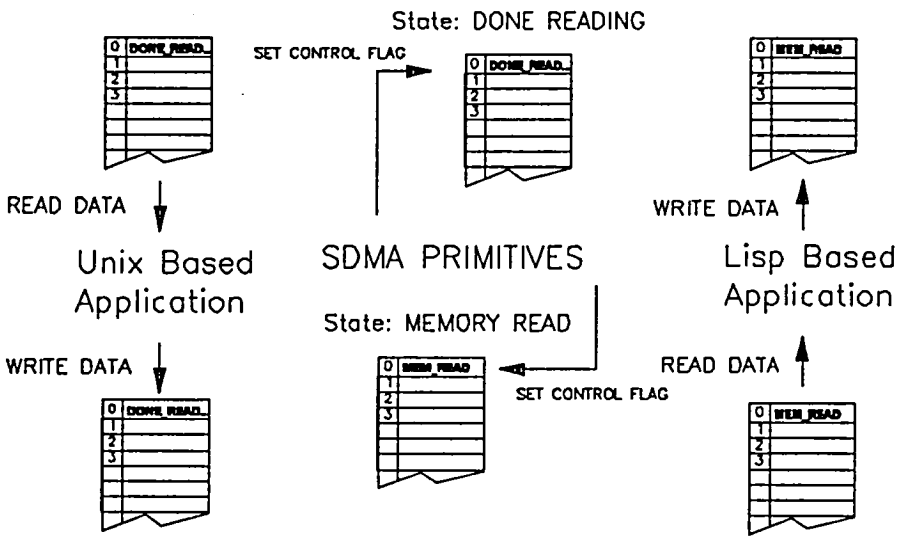

Fig. 6 Read/Write Mode Cycle

Read/Write Mode:

A Unix based application and a Lisp based application have sequential data processing requirements as shown in Fig. 6. In other words, a Unix application needs data from Lisp in order to generate data needed by Lisp to generate data needed by Unix etc. Here, the Unix based application is arbitrarily given responsibility for termination.

III. Usage

The shared memory primitives on the Unix and Lisp sides are written with the view of the Explorer as the master and the M68020 as the slave. The reason for this perspective is a direct result of the flexible window environment and user interface supported on the Explorer. Typically, applications will be initiated from a command menu in a window-based environment for the application and the primitives are designed in a fashion consistent with this view.

A. Using the Primitives

Unix Side

On the Unix side, a driver must be supplied for each mode of operation using a prespecified naming convention. In particular, for any given mode, the user is responsible for supplying the routine for his application as well as two dummy programs (stubs) for the other modes which are not used. However, stubs (dummy routines with no statements) can be generated readily from the examples provided with the SDMA primitives.

The following routines must be supplied by the user in order to interface with the Unix side shared memory drivers which have been supplied. In particular, the name of the procedure **must appear exactly** as listed below (in lower case). The vector of information to be shared is passed by reference from the routine via the parameter list. Consequently, the routines do not return a value explicitly (ie. they are not functions).

Procedure

readdriver - User defined driver for the read mode of the shared memory. This procedure generates a vector of data of some specified length for use by the Lisp side for each invocation.

Parameters

vector - Array of 32 bit floating point numbers of length MAX_DATA which contains the shared data.

num_pts - Maximum number of memory cells requested to share the data.

Procedure

writedriver - User defined driver for the write mode of the shared memory. This procedure takes a vector of data supplied by the Lisp side of some specified length and performs any necessary processing for each invocation.

Parameters

vector - Array of 32 bit floating point numbers of length MAX_DATA which contains the shared data.

num_pts - Maximum number of memory cells requested to share the data.

Procedure

readwritedriver - User defined driver for the read/write mode of the shared memory. This procedure takes a vector of some predetermined length from the Lisp side and generates another vector of data for the Lisp side on each invocation.

Parameters

vector - Array of 32 bit floating point numbers of length MAX_DATA which contains the shared data.

num_pts - Maximum number of memory cells required to share the data.

Lisp Side

On the Lisp side, a slightly more general procedure can be defined as a result of the flexibility of the Lisp programming language. In particular, it is not necessary to prespecify the names of the routines for each of the modes associated with the shared memory before run time as done above. As a result, it is much simpler to design a general purpose program for dealing with the various cases of interest.

Procedure

sdma_driver - Top level driver for the shared memory on the Lisp side. All shared memory applications must invoke this procedure to manage the transferral of data between a Unix application and a Lisp application.

Parameters

lisp_simu - Name (symbol) of the Lisp driver for the simulation. Since this is the highest level, the driver does not take any arguments.

unix_simu - String specifying the path, name and arguments of the Unix side driver for the application.

mode - Mode of operation for which the memory will be used. Mode must be one of the following:

READ-ONLY
WRITE-ONLY
READ-WRITE

num_pts - Maximum number of data points to be shared on each invocation of the routine.

Optional Parameters

init-proc - Name (symbol) of a Lisp function which supplies any initialization information required by the Unix simulation before the start of the application for either the READ or READ-WRITE mode of operation.

B. Examples

In the following paragraphs, three simple examples are given which demonstrate the usage of the shared memory primitives. In each case, an array of data of some predetermined length (5 in this case) is generated and stored in shared memory at which point a waiting process retrieves the information for further processing.

In order to test the examples, the SDMA primitives and examples must be loaded from the distribution tape, and a listing of the tape contents may be found in Appendix 1. An installation utility has been written in order to facilitate the loading of files from the distribution tape into the appropriate directories on your Explorer system. To load the SDMA primitives and examples, copy the file "Install.lisp" from the distribution tape to your working directory. Load the file contents by substituting your directory for "working_directory" in the following Lisp form

```
(load "sys:working_directory;install.lisp").
```

The installation utility may be run by issuing the Lisp command

```
(install-sdma).
```

Once the files have been loaded from the distribution tape, the SDMA system and package must be created by executing the form

(make-system 'SDMA).

On the Lisp side, the SDMA primitives and examples are located in the directory "sys:public.sdma". The examples may be run by evaluating the lisp forms

(sdma:run1) -- example1
(sdma:run2) -- example2
(sdma:run3) -- example3

for each of the three examples to be discussed below.

On the Unix side, both Fortran and "C" source code for the examples have been provided and may be found the files "/sdma/example/fsdma_stubs.f" and "/sdma/example/sdma_stubs.c" respectively. The "C" examples may be generated by a "super user" without modification by executing the Unix command

make c_examples

in the "/sdma/example" directory. If "super user" status is not obtainable, the sdma directory must be copied to the user's directory and the appropriate pathnames changed in the makefile. The Fortran examples may be generated in a similar manner (ie. make for_examples).

All macros for the SDMA are defined in the include file "/usr/include/smem_dat.h". The source for the open and close operations for the SDMA may be found in "/usr/include/smem.h" and follow the code given in [1].

Example1 -- Read Mode

This example demonstrates the usage of the read mode option. In this simple example, an array of some specified length is generated on the Unix side and then written out to memory to be read by the waiting Lisp application. Lisp reads the data in the memory and then writes the contents to the default data stream. This process is repeated a fixed number of times until the Unix side sends a FINISHED message to terminate the process.

Lisp Code

```
(defun example1 (vector)  ;; Vector contains the shared data!
        "This example takes a vector sent by the Unix side and displays
         it on the TTY.  Test of READ-ONLY mode."

 (print "Information received from Unix!")
 (print vector)
)
```

```
(defun run1 ()  ;; Execute this function to run example one!!
        "Example of Read only mode for SDMA."

  (sdma-driver 'example1 "/sdma/example/c_examples read 5" READ-ONLY 5)
 )      ;; example1 is the name of the Lisp simulation.
        ;; /sdma/example/c_examples is the path and name of the Unix simulation.
        ;; Fortran examples can be executed by substituting for_examples for
        ;; c_examples above.
        ;; The number 5 represents the quantity of shared data.
```

C Code

```
void readdriver (vector,num_pts)
/* Example of a user supplied procedure which writes information
   to the shared memory to be accessed by a Lisp application. */

        float vector[]; /* Shared data from memory. */
        int   *num_pts; /* Must pass by reference to conform with
                           Fortran standard. */
{
        int j;

        if (vector[1] > 15.0)  /* Stopping Criterion */
                vector[0] = (float)FINISHED;
                                /* Send a FINISHED message to
                                   terminate the sharing process. */
        else
                for(j=1; j <= *num_pts; j++)
                        vector[j] = j + (float)vector[*num_pts];
}
```

Example2 -- Write Mode

This example demonstrates the usage of the write mode option by performing the converse of example1. Here, an array of data is generated by Lisp and sent to Unix for processing (e.g. the vector is written to the TTY). One important difference with respect to example1 above is the array printed by the Unix application cannot be viewed explicitly since the program is executed from the Lisp side. As a result, the current output stream is set to the Lisp Listener which cannot be accessed directly from the Unix side.

Lisp Code

```
(defun example2 (vector)  ;; Vector contains the shared data.
        "This example generates the data vectors to be written to the Unix
         side.  Demonstration of the Write only mode."

 (let* ((temp vector))

        (cond ((= count 6)              ;; Quitting criterion
                        (setf (aref temp 0) (coerce FINISHED 'float))
                        temp)           ;; Send a FINISHED message
                                        ;; to terminate the sharing
                                        ;; process.

                (T                      ;; Default
                        (do ((i 0 (1+ i)))
                           ((= i 5))  ;; 5 elements to be shared

                                (setf (aref temp i) (+ i (* 5 count)))
                                        ;; Generate data
                        ) ;; end of do
                 (print "Information sent to Unix!")
                 (print temp)
                 (incf count)
                temp) ;; Return temp to be written to shared memory.
        )
  )
)

(defun run2 ()  ;; Execute this function to run example 2!!
        "Example of Write only mode for SDMA."

  (setf count 0)  ;; Initialize the counter
  (sdma-driver 'example2 "/sdma/example/c_examples write 5" WRITE-ONLY 5)
 )      ;; example2 is the name of the Lisp simulation.
        ;; /sdma/example/c_examples is the path and name of the Unix simulation.
        ;; Fortran examples can be executed by substituting for_examples for
        ;; c_examples above.
        ;; The number 5 represents the quantity of shared data.
```

C Code

```
void writedriver (vector,num_pts)
/* Example of a user supplied procedure which reads data written
   to the shared memory by a Lisp application. */
        float vector[]; /* Vector of shared data. */
        int   *num_pts; /* Number of data pts. to be shared. */
{
        int i;

        for(i=1; i <= *num_pts; i++) printf("%f\n",vector[i]);
}
```

Example3 -- Read/Write Mode

The following is a simple example of an application of the read/write mode of the SDMA. A vector of data is generated by the Unix side and passed to shared memory. The memory is read by a Lisp application where its contents are written to the standard output. Once displayed the data are incremented by some fixed amount and then displayed again before being written back to shared memory for the waiting Unix application. This process repeats itself for a fixed number of times before a FINISHED message is sent to Lisp via the Unix application.

Lisp Code

```
(defun example3 (vector) ;; Vector contains the shared data.
        "This example is a test for the read/write mode for SDMA."

  (print "Information sent to Lisp!")
  (print vector)

  (let* ((temp vector))

        (do ((i 0) (1+ i))
           ((= i 5))                                    ;; Stopping criterion
                (setf (aref temp i) (+ (aref temp i) 5));; Gen new vector.
        )
        (print "Information sent to Unix!")
        (print temp)
        temp)           ;; Example3 returns temp to be written to shared memory.
)
```

```
(defun run3 ()                  ;; Execute this function to run example3
        "Example of read/write mode for SDMA."

  (sdma-driver 'example3 "/sdma/example/c_examples rdwrite 5" READ-WRITE 5)
 )      ;; example3 is the name of the Lisp simulation.
        ;; /sdma/example/c_examples is the path and name of the Unix simulation.
        ;; Fortran examples can be run by substituting for_examples for
        ;; c_examples above.
        ;; The number 5 represents the quantity of shared data.
```

C Code

```
void readwritedriver(vector,num_pts)
/*This is an example of a simple application of the read/write mode
  of the SDMA.*/
        float vector[]; /* Vector of data to be shared. */
        int   *num_pts; /* Number of data pts. in memory. */
{
        int i;
        void write_vector(); /* Write *num_pts contents of vector
                                to the TTY. */

        if (vector[1] > 35.0) /* Stopping Criterion */
                vector[0] = (float)FINISHED;
                                /* Send Finished message to terminate
                                   the sharing process. */
        else
        {
                printf("Information sent to Unix\n");
                write_vector(vector,*num_pts);
                for(i=1;i <= *num_pts; i++)
                        vector[i] = (float)i + (float)vector[*num_pts];
                printf("Information from Unix to Lisp\n");
                write_vector(vector,*num_pts);
        }
}
```

C. Building an Application

Once the design and coding of your particular application on both the Unix and Lisp sides has been completed, you must combine the shared memory primitives with your application in order to create an executable program.

1. Lisp Side

In order to use the primitives in your application, make the SDMA system as shown above. Once the SDMA system has been created, the "sdma-driver" may be accessed in manner consistent with the protocol demonstrated in the examples above. Keep in mind that the Lisp function you supply as an argument to the "sdma-driver" must return a vector of data to be written to shared memory if either "write" or "rdwrite" mode is used.

2. Unix Side

On the Unix side, your application must be linked along with the Fortran or "C" driver and written to an executable file in your work directory. The name of the executable file is used by the Lisp side driver for starting the simulation by issuing a shell command along with the appropriate arguments. In an attempt to ease the integration effort on the part of the programmer, the "make" utility is used for building the Unix-based application with the SDMA primitives.

The following steps must be performed for the final integration:

Step:

1. Copy the makefile supplied with the SDMA examples into your work directory with the command

   cp /sdma/example/makefile "work directory"

   where "work directory" is the appropriate pathname.

2. Modify the makefile to suit your specific application by specifying the names of the simulation drivers in the following manner:

   C_OBJECTS = "List of all *.o files used in your simulation.",

   or if your simulation is written in Fortran then

   F_FILES = "List of all *.f files used in the simulation."

   F_OBJECTS = "List of all corresponding *.o files."

3. Run "make" in your work directory to generate executable code for the Unix based application. Make sure you have changed the name of the executable file to a name consistent with usage in the Lisp side simulation driver. For example:

make c_examples

will make the executable file "c_examples" for the "C" examples associated with the SDMA in your work directory. The following would be used to make the executable file "for_examples".

make for_examples.

See the manual on Programmer Tools for further discussion of the "make" utility.

D. Making Changes

Typical changes to the SDMA primitives as supplied involve changing the default data type used for the memory or altering the size of the memory block. The alterations can be made very simply by changing the Macros located in the include file "/usr/include/smem_data.h".

Other changes involving the basic structure and/or operation of the primitives can be achieved by altering the supplied source code. The Lisp source is readily accessible as mentioned above. The Unix side source is contained in the file "/sdma/sdma_driver.c". Source code for the opening and closing of the shared block of memory can be found in "/usr/include/smem.h".

If any changes are made to the include files associated with the primitives, the make utility must be executed in order incorporate the alterations into the current build of the Unix based application.

APPENDIX 1

Contents of the distribution tape.

Install.lisp
defsystem.lisp
SDMA.system
SDMA-driver.lisp
SDMA-driver.xld
SDMA-examples.lisp
SDMA-examples.xld
c_sdma.c
fsdma_driver.f
sdma_driver.c
fsdma_stubs.f
makefile
sdma_stubs.c
smem_dat.h
smem.h

APPENDIX 2

** Procedure Design Specification for SDMA primitives on the
** Unix side of the TI Explorer II-LX

Macros:

MAX_DATA	100	--	Maximum number of data blocks in the shared memory.
MEM_READ	1	--	Flag indicating the receiving process has accessed the information provided by the sending process.
DONE_READING	2	--	Flag indicating the receiving process has NOT accessed the information provided by the sending process.
FINISHED	3	--	Flag indicating the sending of information across the shared memory is complete.
QUIT	0xFFFF	--	Quit flag used to terminate the data transfer procedures.
READ_ONLY	0	--	Indication of read-only mode. (ie. The Lisp side is reading data from the Unix side.)
WRITE_ONLY	1	--	Indication of the write-only mode. (ie. The Lisp side is writing data to the unix side.)
READ/WRITE	2	--	Indication of the read/write mode.
NOCACHE	0	--	Turn the CACHE memory off.
CACHE	1	--	Turn the CACHE memory on.

PAGES btopgs(MAX_DATA * sizeof(float))
-- Number of pages of S1500 memory to be wired down for the given application. Note: 32 bit words are the accepted medium of data transfer. If more accuracy is required, then another data type must be substituted for "float".

****

Procedure: Setup_SDMA

Purpose:

This procedure takes the virtual address of a data array of length MAX and builds a data structure which may be accessed by the Lisp side for the transfer of data. Basic error checking is performed on the wired down memory in order to ensure that no errors will occur during the data transfer.

Inputs:

mode -- mode for which the memory will be used. In particular, three modes are defined:
READ_ONLY
WRITE_ONLY
READ/WRITE.

data -- address of the memory used for data transfer.

Outputs:

success -- return a success flag indicating the status of the attempted operation.

Local Data:

command -- a structure of type SDMA which is predefined in smem.h and used to hold the data required to perform the shared memory operations.

sdma_fd -- an integer which acts as a descriptor for the virtual memory to be accessed by the Unix system.

ds_fd -- an integer which acts as a device descriptor for the logical device used to access the shared memory by System V Unix.

sdmaseg_id -- an integer representing the memory segment identification number.

num_pages -- an integer indicating the number of required pages of S1500 memory that exist whenever the ioctl procedure call is made.

paddr -- array of length PAGES containing the address of each page of wired down S1500 memory.

Calls:

BTOPGS -- macro which does not require the beginning address of the memory segment in order to determine the memory requirements. This macro returns an integer representing the number of pages necessary for the worst possible page alignment of the data array. Note: this macro is different from btopgs (See LX User Manual for a more detailed discussion).

open -- "C" primitive which opens an external device for read-only, write-only, or read/write access.

ioctl -- Input/Output control function which provides an interface to the SYSTEM V kernel. Three possible actions may be performed and are outlined as follows:

SIOCGET - finds the number of currently available pages of S1500 memory to be wired down.

SIOCSET - locks down the requested pages of S1500 memory in order to ensure that it is not used by any other process utilizing blocks of S1500 memory. This option requires the last argument of ioctl to be instantiated to a cmd data structure as defined in "smem.h".

SIOCREL - release the locked down pages of S1500 memory. The last argument of ioctl is an integer that identifies the segment of memory to be unlocked.

Method:

"Initialize the SDMA command data structure as follows:"
Set the .vaddr (virtual address) to the address of the data array,
Set the .segsz (segment size) to the number of PAGES required by the data array,
Set the .cache flag to NOCACHE,
"Having the cache turned off will degrade the performance of the shared memory procedure. Hence, the cache should be turned off only when data integrity becomes a serious concern given various computational requirements."

Set the .frame to point to paddr which contains the base physical addresses for the pages of shared memory.

"Open the shared memory device for Read/Write access. The shared memory is viewed as a logical device by Unix and is opened by the following procedure call."
Set the sdma_fd to open("/dev/sdma",O_RDWR).

"Error Checking."
If the sdma_fd is less than zero (ie. -1) then print an error message, otherwise the open call executed properly.

"Determine the number of pages which is currently available to be locked down by the caller."
Set num_pages (number of pages of wired down memory) to ioctl(sdma_fd, SIOCGET, 0).

"Error Checking."
If num_pages is less then PAGES, then a sufficient amount of S1500 memory does not exist for the requested data transfer. MAX must decrease or other administrative action must be taken in order to provide the necessary memory. Notify user via a message sent to the terminal and kill the process.

"Lock down the necessary memory!"
If num_pages is greater then PAGES then
set sdmaseg_id to ioctl(sdma_fd, SIOCSET, &command)
which will return the S1500 memory segment ID based
on the cmd (command) data structure defined earlier.

"Open the data stream defined by the user (ie. ds_83)."
Set ds_fd to open("/dev/ds83", O_RDWR) where a file
descriptor ID is returned if the device could be opened
for READ/WRITE access and a -1 otherwise.

"Error Checking."
If the ds_fd is less then zero than an error condition
has been observed. Hence, an appropriate error message
must be written and the process terminated.

"Write the information in the command structure to the
initialized data stream."
write(ds_fd, &command, 8)

"Attach the physical addresses of the wired down memory
in the paddr array to the data stream."
write(ds_fd, paddr, PAGES * sizeof(float))

"Close the logical device."
close(ds_fd)
"The last two elements in the data array are reserved
to act as registers containing information about the
segment of S1500 memory."
Set data[MAX-2] to the sdma_fd.
Set data[MAX-1] to the sdmaseg_id.

Return success to caller since the SDMA has been setup.

****

Procedure: Close_SDMA

Purpose:

This procedure uses the last two elements of the data array
which contain the device file descriptor and the ID of the
shared memory and unlocks the wired down memory. The shared
memory is restored to the system heap and the process terminated.

Inputs:

data	--	array containing the necessary administrative information to unlock the wired down memory of length MAX_DATA.

Outputs:

success	--	flag indicating the SDMA has been closed properly.

Local Data:

None.

Calls:

ioctl	--	Input/Output control function which provides an interface to the SYSTEM V kernel.

Method:

"Wait until the Lisp side has sent a quit message
indicating a termination of the Lisp side driver."
While data[0] <> QUIT do nothing until the condition is true.

"Release the wired down S1500 memory and terminate the process."
If ioctl(data[MAX-2], SIOCREL, data[MAX-1]) < 0 then
print an error message notifying the user that
the memory could not be unlocked and return NOT success.

Else return success.

****

Procedure: Sdma_Driver

Purpose:

This procedure is the driver for the shared memory
primitives for System V Unix. The options are passed
to this procedure via the command line and branches are made
based upon the indicated mode.

** Note: special considerations are necessary if the implementation of this routine is done in Fortran. In particular, a call will be made to get_arg and to a SDMA_DRIVER written in C which handles the method outlined below.

Inputs:

mode -- The mode of operation for the shared memory. In particular, modes are entered via the command line in lower case as follows:

read
write
rdwrite.

num_pts -- The maximum number of data points to be passed across the shared memory. This information is provided via a command line argument as well.

unix_simu -- Driver for the Unix process with functionality based upon the selected mode of operation. The Unix process must conform to the established naming conventions and be linked in with the final build. The naming conventions for all three modes are as follows:

readdriver	--	READ MODE
writedirver	--	WRITE MODE
readwritedriver	--	READ/WRITE MODE.

Outputs:

None. (Top level driver)

Local Data:

None.

Calls:

read_mode -- The Lisp side is reading from Unix.

write_mode -- The Lisp side is writing to Unix.

read_write_mode -- Two way data transfer is required over the same address space in wired down memory.

Method:

"Branch on the mode of operation. The mode and the number of data points are identified via the array argv[] as specified for the Unix System V kernel.

If the mode is "read" then
If Not read_mode(unix_simu,num_pts) then notify the user via a print message that an error has occurred.

Else if the mode is "write" then
If Not write_mode(unix_simu,num_pts) then notify the user via a print message that an error has occurred.

Else if the mode is "rdwrite" then
If Not read_write_mode(unix_simu,num_pts) then notify the user via a print message that an error has occurred.

Else the mode is unknown and the user is notified with an appropriate error message.

****

Procedure: read_mode

Purpose:

Perform the necessary operations required to set up the shared memory and establish the communication protocol for passing information from the Unix side to the Lisp side via the shared memory. This level is invariant to the particular unix_simu used for generating data for use on the Lisp side. In read mode, the Unix side will terminate the process.

Inputs:

unix_simu -- Name of the function which will generate the data to be passed to the Lisp side.

num_pts -- Number of data points to be generated upon each consecutive call to the unix_simu. Note: num_pts ≤ MAX_DATA - 3. (num_pts must be declared in the same location as the unix_simu.)

Outputs:

success -- True, if the operation was successful and False otherwise.

Local Data:

vector -- Array of length MAX_DATA of type float which is the chosen data interface for the shared memory used in the current set of applications.

buffer -- Array of length MAX_DATA of type float which is used to hold the data generated by the Unix side until it is ready to be sent to the Lisp side.

Calls:

setup_sdma -- Lock down the memory needed.

unix_simu -- Address of the function which will supply information to the Lisp side.

close_sdma -- Return the wired down memory to the heap for reallocation.

Method:

"Set up the shared memory."
If setup_sdma(vector) then begin passing information to the Lisp side.

"Wait for the Initialization data to be send by Lisp."
While vector[0] <> DONE_READING do nothing.
"Generate the first vector of data to be read by Lisp using any initialization data which has been supplied."
Set vector to unix_simu(vector,num_pts).

```
        "Pass the data to the buffer for conformity with the
         general looping algorithm."
    Set buffer to vector.

        "Loop until the Unix process sends a FINISHED message."
    While vector[0] <> FINISHED do

            "Tell Lisp that new information is ready to be read."
        Set vector[0] to MEM_READ.

            "Generate a new buffer to be sent to Lisp."
        Set buffer to unix_simu(buffer,num_pts).

            "Loop until the Lisp side has read the array of
                data sent by Unix."
        While vector[0] = MEM_READ do nothing.

            "Once the Lisp has read the information in the
             shared memory, the new data may be wired down from
             buffer to vector."
        If vector[0] = DONE_READING Then

                "Transfer the contents of the data buffer to the
                 shared memory.  Note: the Unix side driver must
                 generate a FINISHED message when the process
                 is complete."
            Set vector to buffer.

    "End of the While NOT FINISHED block."

    If close_sdma(vector) then the wired down memory
        has been returned to the system heap.
        return a TRUE flag.

    Else
            "The memory could not be released so the
             procedure must be terminated abnormally.
             Print an error message and exit."
        return a FALSE flag.

Else
        "The set up of the shared memory failed.
         Print an error message and exit."
    return a FALSE flag.
```

****

Procedure: write_mode

Purpose:

Perform the necessary operations required to set up the shared memory and establish the communication protocol for passing information from the Lisp side to the Unix side via the shared memory.

Inputs:

unix_simu	--	Name of the function which will use the data generated by the Lisp side.
num_pts	--	Number of data points required for each call to the unix_simu. Note: num_pts <= MAX_DATA - 3.

Outputs:

success	--	True, if the operation was successful and false otherwise.

Local Data:

vector	--	Array of length MAX_DATA of type float which is the chosen data interface for the shared memory used in the current set of applications.

Calls:

setup_sdma	--	Lock down the memory needed.
unix_simu	--	Address of the function which will use the information from the Lisp side.
close_sdma	--	Return the wired down memory to the heap for reallocation.

Method:

```
        "Set up the shared memory."
    If setup_sdma(vector) then begin passing information to the Lisp side.

            "Notify the Lisp side that the Unix side is ready to
             begin receiving data."
        Set vector[0] to DONE_READING.

            "Loop until the Lisp side sends a QUIT message
             to terminate the process."
        While vector[0] <> QUIT do the following

                "Once Lisp has sent the data, read it
                 and pass it on to the unix_simu."
            If vector[0] = MEM_READ then

                    "Process the data using the provided Unix driver."
                unix_simu(vector,num_pts).

                    "Free the Lisp side to write a new vector."
                Set vector[0] to DONE_READING to indicate the
                    information has been read by Unix.

            "End of the MEM_READ block."

                "If the Lisp side has not sent down new information
                 then simply loop until information is in the
                 shared memory or a FINISHED message is received."

            If close_sdma(vector) then the wired down memory
                has been returned to the system heap.
                return(TRUE).

            Else
                    "The memory could not be released so the
                     procedure must be terminated abnormally, so
                     print an error message and exit."
                return(FALSE).

    Else
            "The set up of the shared memory failed.
             Print an error message and exit."
        return(FALSE).
```

****

Procedure: read_write_mode

Purpose:

Perform the necessary operations required to set up the shared memory and establish the communication protocol for passing information from the Unix side to the Lisp side and back via the shared memory. Unix will send a Finished message when the process has ended.

Inputs:

unix_simu	--	Name of the function which will use data from the Lisp side and generate new data to be used by the Lisp side.
num_pts	--	Number of data points required for each consecutive call to the unix_simu. Note: num_pts <= MAX_DATA - 3.

Outputs:

success	--	True, if the operation was successful and false otherwise.

Local Data:

vector	--	Array of length MAX_DATA of type float which is the chosen data interface for the shared memory used in the current set of applications.

Calls:

setup_sdma	--	Lock down the memory needed.
unix_simu	--	Address of the function which will use the information from the Lisp side and create new data to be used by Lisp.
close_sdma	--	Return the wired down memory to the heap for reallocation.

Method:

"Set up the shared memory."
If setup_sdma(vector) then begin passing information to the Lisp side.

"Wait for initialization data to be sent by Lisp."
While vector[0] <> DONE_READING do nothing.

"Generate the first buffer to be sent to Lisp
using any of the supplied initialization data."
Set vector to unix_simu(vector,num_pts).

"Loop until the Unix process sends a FINISHED message."
While vector[0] <> FINISHED do

"Tell the Lisp side that the data is ready."
Set vector[0] to MEM_READ.

"Loop until the Lisp side has read the
array of data and sent back new data."
While vector[0] = MEM_READ do nothing.

"Generate a new buffer to be sent to Lisp."
Set vector to unix_simu(vector,num_pts).

"End of the while loop."

If close_sdma(vector) then the wired down memory
has been returned to the system heap.
return(TRUE).

Else
"The memory could not be released so the
procedure must be terminated abnormally.
Print an error message and exit."
return(FALSE).

Else
"The set up of the shared memory failed.
Print an error message and exit."
return(FALSE).

==

** Procedure Design Specifications of the SDMA Procedures for
** the Lisp side of the TI Explorer II-LX.

Global Data

LX:LX-SDMA-DATA-STREAM-ID	#x83	Name of the shared memory direct data stream.
QUIT	#xFFFF	Quit flag sent by the Unix side when the simulation has terminated.
MAX-DATA	100	Maximum number of data elements to be passed via the shared memory. NOTE: this includes the control values, hence the actual number of data cells is MAX-DATA - 3.
MEM-READ	1	Read the data in shared memory.
DONE-READING	2	The data in the memory has been read.
FINISHED	3	The information passing process is complete and processes on both sides are permitted to end.
READ-ONLY	0	Flag for read only usage.
WRITE-ONLY	1	Flag for write only usage.
READ-WRITE	2	Flag for read/write usage.

****

Procedure: SDMA_DRIVER

Purpose:

This procedure is the driver for the shared memory procedure on the TI Explorer. All process drivers etc. are called from this routine based upon the given mode of operation.

Inputs:

lisp-simu	--	name of the process function on the Lisp side which drives the entire simulation package developed and supplied by the user. (Note: this function must take the list of data written by the appropriate routine on the Unix side as an argument.)
unix-simu	--	string specifying the path and name of the Unix side process driver as defined above. Included with the name of the driver is a list of command line arguments which includes the mode of operation and the number of data points to be passed.
mode	--	indication of the mode of operation, ie. READ-ONLY, WRITE-ONLY, or READ-WRITE.
num-data	--	the total number of array elements to be shared for a given mode of operation.

Optional Inputs:

init-proc	--	routine which supplies any initialization data required by the Unix process.

Outputs:

None.

Local Data:

data-stream	--	defines the direct data stream over which the shared information will pass.
lx-vm-map	--	virtual to physical memory map used in the creation of the shared memory.
lx-addr	--	starting address of the virtual memory map.
info-vector	--	vector of type float of length num-data which is used to hold the information loaded from the Unix side.

Calls:

lx:open-direct-stream	--	open a direct stream between the Lisp side and the Unix side on the TI Explorer which is uniquely specified by the device number passed as an argument.
lx:make-lx-vm-map	--	create a map of S1500 virtual to physical memory address translations and store it on the Lisp side.
lx:lx-vm-map-vaddr	--	access the memory map created on the Lisp side and return the beginning virtual memory address.
lx:issue-shell-command	--	issue a TI System V command specified by a string argument.
unwind-protect	--	special form which protects against nonlocal exits such as an abort issued by the user. This form allows the user to specify a set of functions to be executed if a nonlocal exit occurs giving control over error conditions to the programmer.

lx:store-addr	--	write a prespecified number of bytes from the Lisp side to the S1500 side.
lx:load-disp	--	read a prespecified number of bytes from the S1500 side to the Lisp side.
lx:store-vector-disp	--	write a vector of data from the Lisp side to the shared memory to be read from the Unix side.
lx:load-vector-disp	--	read a vector of bytes of specified length from the shared memory.

Method:

"Create the default direct stream on the Unix side."
lx:issue-shell-command("mknod /dev/ds83 c 5 6291587") which creates a direct stream, ds83, which passes characters and may be referenced via a hexadecimal identifier as #x600083 or in short form as #x83.

"Initiate the Unix side driver via a shell command."
lx:issue-shell-command(unix-simu) which will start the process on the Unix side.

"Open a direct stream for bidirectional data transfer."
Set data-stream to
lx:open-direct-stream(LX:LX-SDMA-DATA-STREAM-ID,:bidirectional) which will open the stream referenced by #x83 for bidirectional usage.

"Make the virtual to physical memory map used to access the shared memory and attach to the direct stream."
Set lx-vm-map to lx:make-lx-vm-map(data-stream).

"Access the memory map and return the beginning address of the virtual memory."
Set lx-addr to lx:lx-vm-map-vaddr(lx-vm-map).

"The shared memory has been created and all necessary information has been obtained."

"Protect the following from external exits by using the unwind-protect procedure."

*** unwind-protect ***

If the mode is READ-ONLY then
"READ data from the Unix side and use it on the Lisp side."

if init-proc then
lx:load-vector-disp(:float,4,lx-vm-map,apply(init-proc())) otherwise
lx:load-vector-disp(:float,4,lx-vm-map,info-vector).

do until "The Unix side has sent a FINISHED message."
lx:load-disp(:float,0,lx-vm-map) equals FINISHED
Then lx:load-disp(:float,lx-addr,lx-vm-map,QUIT) and
send the data-stream a close message.

"If the information has not previously
been read by the Lisp side then read it."
If lx:load-disp(:float,0,lx-vm-map) equals MEM-READ Then

"Read the vector of information written to the SDMA by the Unix side."
lx:load-vector-disp(:float,4,lx-vm-map,info-vector)
which loads the vector from the SDMA into info-vector.

"Indicate the information has been read."
lx:store-addr(:float,lx-addr,lx-vm-map,DONE-READING)

"Pass the information to the Lisp side
driver which uses the information."
apply(lisp-simu,info-vector) executes
the lisp-simu given the info-vector as an argument.

"If the information has previously been read by
the Lisp side then loop until the Unix side has
written new information."

** End of the do loop for the READ-ONLY mode. **

If the mode is WRITE-ONLY
"WRITE data from the Lisp side and use it on the Unix
side. Note: the Unix side must know how many words of
data are to be written by the Lisp side. This is the
responsibility of the Unix side drivers."

```
    "Get the information from the Lisp side."
  Set info-vector to apply(Lisp-simu,info-vector) which
       returns the information required by the Unix side.

  do until  "The FINISHED message comes from the Lisp
             side and the Unix side is done reading the
             data sent by the Lisp side."
        the first element in info-vector equals FINISHED
             AND DONE-READING equals lx:load-disp(:float,0,lx-vm-map)
             Then send the data-stream a close message,
                  and exit the do loop.

             "If the information has been read by Unix
              then write new information and reset the flag."
        If lx:load-disp(:float,0,lx-vm-map) equals
                  DONE-READING and a FINISHED message has not been
                  received by the Lisp simulation routine then

              "Write the information vector to Unix."
             lx:store-vector-disp(:float,4,lx-vm-map,info-vector)
                  which stores the contents of info-vector
                  in the shared memory.

              "Reset the read flag for Unix."

             lx:store-addr(:float,lx-addr,lx-vm-map,MEM-READ)

              "Get a new information vector!"
             Set info-vector to apply(Lisp-simu,info-vector).

        "If the information has not yet been read by the
         Unix side, then loop until the data has been accessed."

  ** End of do loop for WRITE-ONLY mode. **

If the mode is READ-WRITE Then
  "Data is sent both ways through the shared memory."

  If init-proc Then
       lx:load-vector-disp(:float,4,lx-vm-map,apply(init-proc())) otherwise
       lx:load-vector-disp(:float,4,lx-vm-map,info-vector).

       "The quit message comes from the Unix side just
        as in the READ-ONLY mode presented above."
```

```
do until
    lx:load-disp(:float,0,lx-vm-map) equals FINISHED
        Then send the data-stream a CLOSE message
        and exit the do loop.

        "If the information has not previously
         been read by the Lisp side then read it."
    If lx:load-disp(:float,0,lx-vm-map) equals MEM-READ Then

            "Read the vector of information."
        lx:load-vector-disp(:float,4,lx-vm-map,info-vector)
            which loads the contents of the SDMA memory into the info-vector.

            "Pass the information to the Lisp side driver and
             get new information to pass back to the Unix process."
        Set info-vector to apply(Lisp-simu,info-vector).

            "Write the information back to the Unix side."
        lx:store-vector-disp(:float,4,lx-vm-map,info-vector)
            which stores the new information in the
            shared memory to be read by the Unix side."

            "Indicate completion of the information
             processing performed by the Lisp side.
             This is an signal for the Unix process to perform
             the necessary operations on the supplied data."
        lx:store-addr(:float,lx-addr,lx-vm-map,DONE-READING)

    "If the information has not been processed and
     written by the Unix side for the Lisp process,
     then loop until the data has been transferred
     or a FINISHED message is received."

** End of the do loop for READ-WRITE mode. **

"The clean-up functions for the unwind protect include,
 sending a QUIT message to the Unix process and closing
 the data stream used to send data to the shared memory."

lx:store-addr(:float,lx-addr,lx-vm-map,QUIT) which will
    write a QUIT message to the data buffer.
Send the data-stream a close message.

** End of the unwind-protect form. **
```

BIBLIOGRAPHY

[1] "Explorer LX™ User's Guide," Texas Instruments Inc.
Data Systems Group, Austin Texas, Dec 1986, Rev. A. July 1987.

INDEX

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-103289		

4. Title and Subtitle	5. Report Date
Shared Direct Memory Access on the Explorer II-LX	September 1990
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Jeffrey L. Musgrave	E-5747
	10. Work Unit No.
	582-01-11

9. Performing Organization Name and Address	11. Contract or Grant No.
National Aeronautics and Space Administration Lewis Research Center Cleveland, Ohio 44135-3191	
	13. Type of Report and Period Covered
	Technical Memorandum

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
National Aeronautics and Space Administration Washington, D.C. 20546-0001	

15. Supplementary Notes

16. Abstract

Advances in Expert System technology and Artificial Intelligence have provided a framework for applying automated "Intelligence" to the solution of problems which were generally perceived as intractable using more classical approaches. As a result, hybrid architectures and parallel processing capability have become more common in computing environments. The Texas Instruments Explorer II-LX is an example of a machine which combines a symbolic processing environment, and a computationally oriented environment in a single chassis for integrated problem solutions. This user's manual is an attempt to make these capabilities more accessible to a wider range of engineers and programmers with problems well suited to solution in such an environment.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Parallel computing TI Explorer II-LX Multiprocessor environment	Unclassified – Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price*
Unclassified	Unclassified	47	A03

*For sale by the National Technical Information Service, Springfield, Virginia 22161

National Aeronautics and
Space Administration

Lewis Research Center
Cleveland, Ohio 44135

Official Business
Penalty for Private Use $300

FOURTH CLASS MAIL

ADDRESS CORRECTION REQUESTED

U.S.MAIL

Postage and Fees Paid
National Aeronautics and
Space Administration
NASA-451

NASA